Keep The Change

Written by Steve Berczuk, Brad Appleton, and Robert Cowham

Monday, 18 May 2009

Software Development teams deliver value by writing new code, or by changing existing code and functionality. The majority of the time it is about changing existing systems, not writing completely new ones. Software Configuration Management is one of the disciplines responsible for managing that change. Rapid, unmanaged, change can cause chaos and reduce productivity. One (easy) way to reduce that chaos is to reduce the frequency of change. We have most of us experienced situations where order has been been created out of chaos by slowing things down and controlling changes such that they are perhaps grouped together (in releases), reviewed and authorised before being allowed to happen, and similar practices.

While it may seem that slowing things down is undesirable, such approaches often work very well, and produce a higher throughput than the original (seemingly more active and energetic - if perhaps rather more frenetic) process. The downside of these experiences is that they make it easy to confuse "manage" with "control", and even easier to interpret "control" as "inhibit." We believe that rapid, well-managed change is a mechanism for delivering value more quickly. Rather than simply managing change, SCM can be a tool for enabling change that maximizes the value of the system being developed.

A traditional SCM approach to manage change is to isolate streams of work to reduce the immediate impact of changes in one stream on others. While this provides for short term gains, it defers the cost of integration til a later, perhaps more sensitive, time in the project. By realizing that branching is not the only tool teams can us to manage change, and by using simple codelines, and good engineering practices, your team can develop new features in an integrated manner, isolating streams of development, and not incurring the associated cost of branching and merging.

Applying the Principles of Lean development to Branching

The principles of Lean development, as applied to Branching are [LEANBRANCH]:

- Eliminate Waste - Eliminate avoidable merge-propagation, long-lived variant branches, and stale code and partially completed work
- Build Quality In - Maintain codeline integrity with an automated unit & integration tests and a Codeline Policy to establish a set of invariant conditions that all checkins/commits to the codeline must preserve.
- Amplify Learning - Facilitate frequent feedback via frequent/continuous integration and workspace update
- Defer Commitment - Branch as late as possible! Create a label to take a "snapshot" of where you MIGHT have to branch from, but don't actually create the branch until parallelism is needed.
- Deliver Fast - Complete and commit change-tasks and short-lived branches (such as task-branches, private-branches, and release-prep branches) as early as possible
- Respect People - Let developers reconcile merges and commit their own changes (as opposed to some "dedicated integrator/builder")
- Optimize the "Whole" - When/if branches are created, use the Mainline pattern to maintain a "leaner" and more manageable branching structure

In other words, branch when it makes sense, but realize that branching has a cost and remember that branching is not always the best tool for managing change.

If you favor eliminating waste, and the other principles of lean branching, working on a single codeline, when in the past you might

have favored a branch, requires a commitment on the part of the development team to maintaining the simplicity necessary to make this process work.

Simple Codelines and No Broken Windows
For teams to make progress, you need to keep the codeline functional. In the words of the Pragmatic Programmers, "clean, functional systems deteriorate pretty quickly once windows start breaking" [ENTROPY]. If the codeline is always working, it's easy to migrate new functionaliy in using adapters, or to add modules for new functionality without integrating them tightly into the existing code. One way of viewing this is as simple codelines. The Rules for simple codelines are [SIMPLE]:

- Correctly build, run (and pass) all the tests
- Contain no duplicate work/work-products
- Transparently contain all the changes we needed to make (and none of the ones we didn't)
- Minimize the number and length of sub-branches and unsynchronized work/changes (see Lean branching)

In the context of simple codelines, branching can still make sense. For example, developers can back their workspaces with Private Branches [SCMPatterns] so that they can checkpoint small tasks. The key for the developer branches to be successful is that they are easy to execute, and that they are isolated from the main development stream for as brief a time as possible.

To maintain a simple codeline, you need to do more than to simpy decree that new work must not break existing code. You need a mechanism to validate that the code is still functional, and you need a plan for incrementally "merging" the new functionality into the application. Automated testing is the best and simplest tool for verifing the stability of the code in the face of change.

Testing and SCM
Some challenge whether testing has a role in SCM. Steve gave a talk at a conference recently, and he saw on the feedback forms more than once a comment to he effect that an attendee was disappointed that the talk covered more on testing and build time integration than on branching. It turns out that testing is a central part of SCM. [SCMTEST]. Consider the definition of CM [DART]:

1. Configuration identification
2. Configuration control
3. Configuration status accounting
4. Configuration audit and review

Automated tests provide the ability to quickly verify that the code performs its desired functionality. Failing tests provide a clue that something has changed and thus a hint that a configuration item may be incorrect. More pedantically you can write tests that validate packaging and version constraints. Automated testing, especially that which runs as part of a continuous integration build, can move you towards these functions by helping you verify that the application's functions are as specified and that any change meets some minimal set of criteria around maintaining existing functionality.

Being able to make changes in an incremental manner requires work on the part of the engineering and architecture teams too, but those who think about SCM can enable this change by providing the infrastructure such and Continuous Integration Environments, testing support, and tools to tie code changes to small stories or requirements that move the project along the way to the new approach incrementally.

Motivating Example
Consider a development team that needs to develop a new valuable piece of functionality that requires a potentially disruptive change to the code (for example, an API change). After looking at the work to be done, and doing some preliminary design work, the team estimates that the work will need more than one person and more than a single iteration to complete. While this functionality is being implemented, other team members will still be working on the active code line.

The team is familiar with Software Configuration Management Patterns [SCMPatterns] and is considering two options (somewhat related) for this team:

- Create a task branch for the project. The team would work in a separate development stream, and when everything is working and they're ready to roll out the new feature, the team would integrate with the main development line.
- Work off the main development line and develop the new features in an incremental manner so that no existing functionaility breaks.

Let's explore the options in turn.

Branching: the good, the bad, and the ugly
A task branch is a traditional SCM solution to handing potentially disruptive change: "Have part of your team perform a disruptive task without forcing the rest of the team to work around them, using a Task Branch."[SCMPatterns] When the work on the task branch is complete, the team merges the task branch back into the active development line so that the new functionality is available for the next release.

Creating a branch has the advantage of isolating the new feature work from the main line work, allowing those working on the task branch to control when to accept changes from the active development line, and allowing those working on the main line to work in blissfull ignorance of the change occuring on the task branch.

Even with the benefits of isolation that branching provides, branching has a bad reputation in some circles. It's considered difficult, and branching sometimes leads to duplication of effort when there is a requirement for a change on both the main development stream and the task branch. In most cases, given modern tools, branching (creating a branch) isn't all that bad. What is problematic about branching is merging changes between branches.

While some tools are better than others, merging still requires an understanding of the intention of the person making the change on both codelines to decide what the right end result is. If the team on the task branch is really doing major restructuring of the code, some changes from the mainline would not make sense to integrate into the branch -- for example, a change to a Java class in the main line might be difficult to reproduce in the task branch if the class disappeared as a result of the refactoring. There are approaches to minimize the risks of a large merge. The Task Branch Pattern recommends that you merge the integrate changes from the main line into the task branch frequently. This eases the difficultly of merging because the amount of code change is less the more frequently you integrate, and the odds of understanding the intent behind the changes on each code line are higher because the activities are more fresh in everyone's memory. Frequent integration does not address the difficulties that will happen as time passes, the codelines diverge, and the odds of understanding how mainline changes fit into a task branch where a major refactoring is going on become less.

When you have development streams that are truly parallel, branches provide the level of isolation needed for parallel work to go on in a stable fashion. A Release Line [SCMPatterns], which is a branch created to support a released product version, is a perfect example of when branching is the right answer for a team. On the release line, there are stricter rules about what can change, and the focus of the release line is on stability, perhaps at the cost of a slower rate of change. Some changes to the release line may need to be effected in the ongoing development stream. Some may not.

Branching, by its nature, increases isolation, and whenever you work on a branch, you are putting off the time when you integrate the changes into the main line. Integration deferred is integration denied in some sense. Sometimes the isolation makes sense, mostly to keep the mainline stable, but if your default way of increasing stability is to branch, your gains may be short-lived, as once you integrate the changes among branches, you may see problems that no one on the team can fully understand the causes of.

A Lean Alternative - "Branching without Branching"
An alternative to a task branch is to work on the active development line and introduce changes incrementally while maintaining a working system. This has the risk of introducing breakage to the codeline if the work is not done carefully, but there are ways to mitigate this risk. The advantage is a simple, lean codeline structure that reduces the effort spent on the integration of changes. In this model you trade effort spent in integrating changes between codelines (which requires work that adds little direct value) with effort spent writing and executing tests that validate that the codeline still works. These tests may require more effort overall to write, but they provide lasting value to the development stream by allowing for continuous validation that the code works.

Continuing to work on the active development line includes quicker feedback about how well the changes integrate with the active

code base, and less overhead to the development team related to merging and propagating changes to the new development stream. Working on one codeline will allow the existing development stream to benefit sooner from the improvements done in service of the task work, and provides more rapid feedback about the reward and risks of the new feature work. This is consistent with the principles of lean branching.

SCM: Change Enabler
An SCM process can use its responsibility for Configuration identification, Configuration control, Configuration status accounting, Configuration authentication as a way to facilitate change or hinder it. By providing the mechanisms for ensuring that the active codeline is evolving in a healthy way, an SCM process can enable a team to deliver more value in a lightweight way. An SCM Process can facilitate change by:

- Providing for a continuous integration process that validates that each change to the code maintains functionality, and notifying the team when something breaks.
- Generate metrics around failing and passing tests, and use these to guide your process.
- Generate metrics around test coverage and use this to verify that new functionality is covered by automated tests and that changes are related to desired functionality.

Branching has a place in your SCM process, but it's not the only tool you have to provide the change management that SCM is responsible for. It's important to remember that change management and change control are not synonymous. By simplifying your codelines and using tools other than branching to manage change, SCM can help agile teams manage change effectively.

References
[SCMTEST] Agile SCM Is testing: http://www.cmcrossroads.com/content/view/9420/264/
[DART] http://www.cmcrossroads.com/cgi-bin/cmwiki/view/CM/SoftwareConfigurationManagement#Susan_Dart
[SCMPatterns] Software Configuration Management Patterns: Effective Teamwork, Practical Integration by Steve Berczuk with Brad Appleton
[SIMPLE] Four Rules for Simple Codelines http://bradapp.blogspot.com/2008/06/four-rules-for-simple-codeline.html
[LEANBRANCH] Lean Branching. http://bradapp.blogspot.com/2006/01/lean-principles-for-branching.html and
[ENTROPY] Software Entropy. http://www.pragprog.com/the-pragmatic-programmer/extracts/software-entropy

---

Brad Appleton *is an enterprise SCM/ALM solution architect for a Fortune 100 technology company. He is co-author of Software Configuration Management Patterns: Effective Teamwork, Practical Integration, the "Agile SCM" column in CMCrossroads.com's CM Journal, and a former section editor for The C++ Report. Since 1987, Brad has extensive experience using, developing, and supporting SCM environments for teams of all shapes and sizes. He holds an M.S. in Software Engineering and a B.S. in Computer Science and Mathematics. You can reach Brad by email at brad@bradapp.net*

Robert Cowham *has been in software development for over 20 years in roles ranging from programming to project management. He continues his involvement in development projects but spends most of his time on SCM Consultancy and Training, now working with Vizim. He is the Chair of the Configuration Management Specialist Group of the British Computer Society, has a BSc in Computer Science from Edinburgh University and is a Chartered Engineer (CEng MBCS CITP). You can contact him at robert@vizim.com .*

Steve Berczuk *is a Technical Lead for an Agile Software Development consulting company. He has been developing software applications since 1989, often as part of geographically distributed teams. In addition to developing software he helps teams use Software Configuration Management effectively in their development process. Steve is co-author of the book Software Configuration Management Patterns: Effective Teamwork, Practical Integration and a Certified ScrumMaster. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact him at steve@berczuk.com*

Hits: 176 Email This Bookmark Set as favorite

Trackback(0)

TrackBack URI for this entry

Comments (0)

Subscribe to this comment's feed

Write comment

Last Updated (Tuesday, 19 May 2009)

Close Window